PROBER: Ad-Hoc Debugging of Extraction and Integration Pipelines

Anish Das Sarma
Yahoo, CA, USA
anishdas@yahoo-inc.com

Alpa Jain
Yahoo, CA, USA
alpa@yahoo-inc.com

Philip Bohannon
Yahoo, CA, USA
plb@yahoo-inc.com

ABSTRACT

Complex information extraction (IE) pipelines assembled by plumbing together off-the-shelf operators, specially customized operators, and operators re-used from other text processing pipelines are becoming an integral component of most text processing frameworks. A critical task faced by the IE pipeline user is to run a post-mortem analysis on the output. Due to the diverse nature of extraction operators (often implemented by independent groups), it is time consuming and error-prone to describe operator semantics formally or operationally to a provenance system.

We introduce the first system that helps IE users analyze pipeline semantics and infer provenance interactively while debugging. This allows the effort to be proportional to the need, and to focus on the portions of the pipeline under the greatest suspicion. We present a generic debugger for running post-execution analysis of any IE pipeline consisting of arbitrary types of operators. We propose an effective provenance model for IE pipelines which captures a variety of operator types, ranging from those for which full or no specifications are available. We present a suite of algorithms to effectively build provenance and facilitate debugging. Finally, we present an extensive experimental study on large-scale real-world extractions from an index of $\sim$500 million Web documents.

1. INTRODUCTION

Growing amounts of knowledge is being made available in the form of unstructured text documents such as, web pages, email, news articles, etc. Information extraction (IE) systems identify structured information (e.g., people names, relations betwen companies, people, locations, etc.) and, not surprisingly, IE systems are becoming a critical first-class operator in a large number of text-processing frameworks. As a concrete example, search engines are moving beyond a "keyword in, document out" paradigm to providing structured information relevant to users' queries (e.g., providing contact information for businesses when user queries involve business names). For this, search engines typically rely on having available large repositories of structured information generated from web pages or query logs using IE systems. With the increasing complexity of IE pipelines, a critical exercise for IE developers and even users is to *debug*, i.e., perform a thorough post-mortem analysis of the output generated by running an entire or partial extraction pipeline. Despite the popularity of IE pipelines, very little attention has been given to building effective ways to trace the control or data flow through an extraction pipeline.

EXAMPLE 1.1. Consider an IE pipeline for extracting contact information for businesses, namely, business name, address (one or many), phone number (one or many), from a set of web pages. The pipeline, in addition to others, consists of operators (a) to clean and parse html web pages, (b) to classify 'blocks' of text in a web page as being useful or not for this task, (c) extract business names, (d) extract address(es). (We discuss this real-world pipeline in detail later in Section 2.) Two interesting points to note here: First, in practice, such complex pipelines may be put together using off-the-shelf operators (e.g., html parsers or segmenters) along with some newly designed as well as some re-usable operators from other systems. Second, IE is an erroneous process and oftentimes, output from an IE pipeline may miss some information (e.g., a record where contact information is present but business name is absent) or may generate unexpected output (e.g., associate a fax number with a business instead of phone number).

Say a user of this IE pipeline processes a batch of web pages and generates a set of (partial, complete, or incorrect) output records. Given the output, the user may be interested in understanding why certain incorrect records were generated to identify and eliminate their 'sources'; similarly, the user may also be interested in understanding why certain records were missing attributes in the output to identify the 'restrictive' operators in the pipeline. □

To date, there have been two main approaches for understanding the output from an IE but neither fully addresses the problem of debugging arbitrary IE pipelines. The first approach is to build statistical models to predict the output quality of an IE system [6, 9]. However, these models address the more modest goal of assessing the overall output quality and lack the intuitive interaction necessarily for building debuggers to trace the generation of an output record. The second approach involves using complete knowledge of how each operator functions. As highlighted by the above example, prior information regarding the specifications of the operators may not be available (e.g., off-the-shelf black-box operators). In the absence of full function specifications of an operator, the only (straightforward) approach to debugging is exploring all data in the pipeline. However, such an approach is clearly infeasible due to the sheer volume of data. For instance, debugging a simple pipeline involving 10 operators with 10,000 input records per operator would require 100K records to be manually examined. (As we shall see in our experiments in Section 6, typical data sizes are even larger.)

This paper presents PROBER (for Provenance-Based Debugger), the first generic framework for debugging information extraction pipelines composed of arbitrary ("black-box") operators. A critical task towards building debuggers is that of tracing and linking output records from each operator and understanding their transformations across different operators in the pipeline. To trace the lineage of any arbitrary record in the output, we propose a novel provenance model for IE pipelines. With debugging in mind, our provenance model tries to minimize the amount of user effort necessary in resolving the fate of the records in the output. For example, provenance for (incorrect) output records *only* refer to input tuples that impacted this output record. We present a suite of algorithms to build the provenance for an IE run; our algorithms explore the tradeoff between efficiency of building provenance, and the amount of information captured by it.

As outlined by Example 1.1, exact functional specifications for

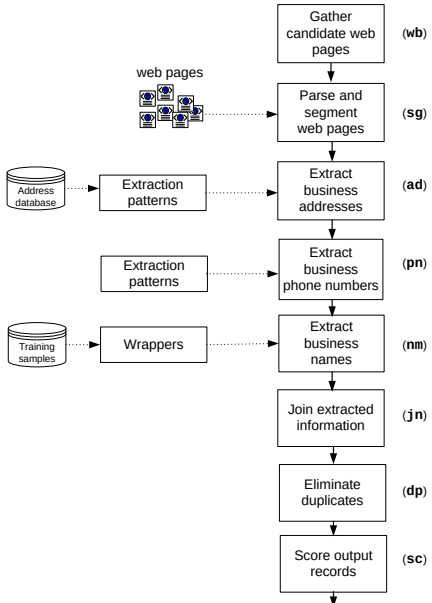


Figure 1: Example of an IE pipeline to generate business names and their contact information.

operators in an IE pipeline may not be available. However, provenance building can exploit various properties of these operators that can be learned by sampling, namely, monotonoic, one-to-one, one-to-many, or arbitrary. We characterize a diverse set of operators, and their properties, that are found in real-world extraction pipelines and rigorously examine methods to build provenance information for each of these combinations.

In summary, beyond the conceptualization of PROBER (Section 2), the main contributions of the paper are as follows:

- A novel provenance model for the task of debugging information extraction pipelines. Our model effectively accounts for scenarios where incomplete (or no) knowledge about the underlying operators in the pipeline is available (Section 3).
- A suite of effective algorithms to build provenance, given an IE run (Section 4).
- An end-to-end solution combining extraction output along with provenance information for debugging (Section 5).
- An extensive evaluation over real-world datasets, demonstrating the effectiveness of our framework in debugging extraction pipelines (Section 6).

2. PROBLEM FORMULATION

While IE pipelines may vary in their implementation logic [1, 10, 12, 13] several underlying common components can be abstracted from the implementation details. We characterize information extraction pipelines for the task of performing post-mortem analysis.

DEFINITION 2.1. **[Record]** A *record* r is a basic unit of data (e.g., a tuple), consisting of a globally unique identifier $I(r)$, and value $V(r)$. We use R denote the set of all records. □

DEFINITION 2.2. **[Operator]** An *operator* is defined by a function $O : (I_1, I_2, \cdots, I_N) \rightarrow R$, where each $I_i \subseteq R$ is a set of records. In practice, the function O may be unknown to us. □

Intuitively, an operator takes as input an N-tuple of sets of records and outputs one set of records. Specifications on how an operator generates an output record may be available in varying forms. Specifically, we consider the following four scenarios involving operator specifications.

An operator is said to be a *black-box* if we have no information about it. In this case, naturally, the only way to gain information about a black-box operator is by executing it on input sets of records. In contrast, we have *exact* information about an operator O if we know precisely *which* input records contributed to each output record, *and how*. We have *Input-Output (IO) specifications* when for each output record, we know which input records were used to construct it, however exactly how a record is generated is unknown to us. Finally, we may have *integrity constraints*, e.g., key-foreign key relationships, satisfied by the input and output records. For instance, an operator may support a 'debug' mode where each output record is assigned an id associated with the input records that generated it. Effectively, using key-foreign keys we have the same information as that in IO specifications, but this information is (indirectly) available using dependencies on the values of *fields* in input and output records.

Next, we define various (standard) properties of an operator, that help design specialized algorithms for building provenance and debugging effectively. As we will see later, these properties may be learned by sampling or the operator specifications (when available) described above.

DEFINITION 2.3. **[Properties]**

- `monotonic`**:** Operator O is `monotonic` iff $\forall I_1, I_2 \subset R : (I_1 \subseteq I_2) \Rightarrow (O(I_1) \subseteq O(I_2))$.
- `one-to-one`**:** Operator O is `one-to-one` iff: (a) $\forall I \subset R : O(I) = \bigcup_{r \in I} O(\{r\})$; (b) $\forall r \in R$: $|O(\{r\})| \leq 1$.
- `one-to-many`**:** Operator O is `one-to-many` iff $\forall I \subset R : O(I) = \bigcup_{r \in I} O(\{r\})$.
- `many-to-one`**:** Operator O is `many-to-one` iff $\forall I \subset R$, $\exists$ a partition $P_I = \{I_1, \ldots, I_n\}$ of I[1] such that: (a) $O(I) = \bigcup_{i=1}^{n} O(I_i)$; and (b) $\forall i : |O(I_i)| \leq 1$.

□

DEFINITION 2.4. **[Extraction Pipeline]** An *extraction pipeline* P is defined by a DAG $G(V, E)$ consisting of a set V of nodes and a set E of edges where each node $v \in V$ corresponds to an operator O in the pipeline. An edge $a \rightarrow b$ between nodes a and b indicates that the output from the operator represented by a is input to operator represented by b. We have a single special node $s \in V$ with no incoming edges representing the operator that takes input to the pipeline, and one special node $t \in V$ with no outgoing edges representing the operator that outputs the final set of records. □

We now discuss a real-world extraction pipeline (used at Yahoo!), which forms the basis of our illustrations in this paper.

Motivating Example

Figure 1 shows a real-world extraction pipeline, `Bussiness`, for building a large collection of businesses (see Example 1.1) by extracting records of the form $\langle n, a, p \rangle$, where business n is located at address a with contact number p. The first step is to build a set of web pages likely to contain information regarding businesses which is done using a variety of document retrieval strategies. Specifically, we issue manually generated domain-dependent queries (e.g., "Toyota car dealership locations") as well as use form filling methods where entries such as, model, make, and zipcode, may be filled in order to fetch a list of car dealerships. This operator, denoted by `wb` is an example of a black-box operator with arbitrary properties.

Given a collection of web pages, operator `sg` parses the html page and identifies appropriate segments of text in this page, where ideally, each segment contains a complete target record (see Figure 4 in the appendix for a real-world example). These segments are then processed by operators, `ad` and `pn`, which respectively

[1](a) $I = \bigcup_{i=1}^{n} I_i$ (b) $\forall i \neq j : (I_i \cap I_j) = \emptyset$.

identify an occurrence of an address and a phone number. The annotation from one operator is used by the subsequent operator to identify regions of text that should not be processed. `ad` and `pn` are implemented using hand-crafted patterns based on a dictionary of address formats. The `nm` operator on the other hand needs to identify names of business which may be arbitrary strings and for this, we follow a wrapper-induction appraoch. In particular, using some training examples we learn a wrapper rule to identify candidate business names; these rules are based on the document structure of the html content. Of course, several other implementations for each of these operators are possible and the implementation details are orthogonal to our discussion since our goal is to build debuggers for pipelines with black-box operators where no implementation information may be available. The `jn` operator joins outupt from `ad`, `pn`, and `nm` to build candidate output records which are, in turn, processed by `dp` to eliminate duplicates. The final operator, assignes a confidence score `sc` to each output record.

We note that all our implementations of the above operators are monotonic. (Obviously, there may be non-monotonic implementations in other pipelines, but we primarily consider monotonic operators in this paper.) Although monotonic, the operators from the pipeline span a variety of properties, e.g., segmentation is a `one-to-many` operation, and by design one address is extracted from each segment, so address extraction is `one-to-one`, while de-duplication is `many-to-one`. Candidate webpage generation and wrapper training, on the other hand are arbitrary, i.e. "`many-to-many`".

Given unexpected output records, an IE developer may want to answer some natural questions about the output. (Figure 4 in the appendix shows an example where `sg` generates an incorrect segment that leads to missing one address and extracting one incorrect address.) Specifically, a developer may be interested in tracing all or part of the input records that contributed to a particular output record. For instance, given an incorrectly extracted record, we would like to know only the relevant subset of webpages and training data that impacted it, i.e., the *minimal* amount of input data necessary to identify the error. Motivated by the above observations, we focus on the following problem in this paper.

PROBLEM 2.1. Given a pipeline P, input I, and partial information about operators in P, we would like to (1) build *provenance* for the set of (intermediate and final) records in the pipeline; (2) expose provenance to developers through a query language and guide them in debugging the pipeline.

3. PROVENANCE FOR IE PIPELINES

The notion of provenance is relatively well-understood for traditional relational databases (refer [5, 17]). A commonly advocated model [2] is to use a boolean-formula provenance, e.g., $S_1 \wedge (S_2 \vee \neg S_3)$. For the purpose of debugging extraction pipelines, such provenance models are not appropriate for two main reasons. First, unlike relational queries where the exact specifications of each operator are known, we may have black-box operators in our extraction pipeline. Second, for debugging, ideally we would like to limit the number of records (and simplify their interdependencies typically represented as boolean formulas for relational operators) a human has to assess in order to understand the issue at hand. With these in mind, we define a provenance model based on *minimal subsets* of operator inputs that capture necessary information (Section 3.1) and extend this basic model to operators where multiple minimal subsets may exist (Section 3.2) .

3.1 MISet: Basic Unit of Provenance

To define the provenance of an IE pipeline P, we begin by defining the provenance for each operator in P; the subsequent sections show how to construct the provenance for each operator in P (Section 4) and for P by composing individual operators' provenance (Section 5). We primarily confine ourselves to extraction pipelines consisting of only monotonic operators (see Definition 2.3), which are a common case in practice (as in our motivating example from Section 2); extensions to non-monotonic operators is very briefly discussed in the appendix (Section C), but largely left as future work.

We define the provenance of an extraction operator O based on the provenance for each output record $r \in R$ for O. Ideally, we would like the provenance of r to represent precisely the set of records that contributed to r, however, as we will see, in practice it may not be possible to always determine the precise set of contributing records (e.g., in the absence of exact information about O), and even if possible it may be computationally intractable. For our goal of building a debugger, we observe that one of the main operations we expect users to perform is look at an (erroneous) output record r, and explore its provenance to determine the cause of this error. Therefore, a suitable provenance model is one that enables users to examine the fewest records required to decide the fate of an output record r. Formally, we define a basic unit of provenance, called MISet as follows:

DEFINITION 3.1. **[MISet]** Given an operator O, its input I and output R, we say that $I_s \subseteq I$ is a *Minimal Subset* (MISet) of $r \in R$ if and only if: (1) $r \in O(I_s)$; and (2) $\forall I' \subset I_s : r \notin O(I')$. We use $M_{all}(O, I, r \in R)$ to denote the set of all MISets of O for input I and output record $r \in R$. □

Intuitively, an MISet gives the fewest input records required for a particular output record r to be present. Therefore, an MISet provides users with one possible reason for the occurrence of r. This, in turn, reduces the burden of manual annotation on the users; in the absence of MISets, a user may have to explore the entire input to understand what caused an error in the output. The notion of MISets primarily focuses on debugging the *presence* of records in the output; in Appendix C we briefly discuss a corresponding notion (MASets) for the case of non-existence of records in the output, but leave details for future work.

In practice, we may have more than one MISet possible for an output record as shown by the following example.

EXAMPLE 3.1. Consider a (simple) record validation operator (e.g., `sc` in Figure 1) that computes the "support" of each record and outputs only records with sufficient support. Suppose `sc` outputs a record r if there are atleast 50 input records supporting it. Given an input of 100 records that could support r, the MISet for r is *any* subset of the input records of exactly 50 records. □

When multiple MISets are available, several ways of building provenance are possible, each differing in the extent to which they impart infomation and execution speed, as explored next.

3.2 Handling Multiple MISets

Several formalisms for provenance model are possible when multiple MISets are available. We rigorously examine compositions of MISets, while capturing the spectrum of complete (and potentially intractable) provenance, to more tractable (but approximate) provenance. Later in Section 4, we will present algorithms for building each of type of provenance.

Consider an operator O which consumes input I and generates output R; for a record $r \in R$, we denote the provenance of r as $P(O, I, r)$. We use subscripts P_* to capture various types of provenance and when clear from the context, we simply use $P_*(r)$ to denote $P_*(O, I, r)$.

Properties	P_{all},P_{imp}	P_{any}	P_{uni}	P_{int}
`arbitrary many-to-one`	#P-complete‡	$\mathcal{O}(MN\alpha^k)^{\dagger}$	$\mathcal{O}(2^N)^{\ddagger}$	$\mathcal{O}(MN)$
`one-to-many`	$\mathcal{O}(M+N)$	$\mathcal{O}(M+N)$	$\mathcal{O}(M+N)$	$\mathcal{O}(M+N)$
`one-to-one unique MISet`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$

Table 1: Complexity of our algorithms for obtaining P_{all}, P_{any}, P_{uni}, P_{int}, and P_{imp} for various properties of an operator O with input I and output O ($N = |I|$ and $M = |O|$). † denotes the number of MISets that need to be found, and α is the size of the largest MISet ($\alpha \leq N$). ‡ denotes cases where complexity becomes PTIME when restricted to bounded-size MISets.

All- and Any-provenance: Ideally for any output record, we would like to provide all possible information using MISets, i.e., capture all possible "causes" of an output record.

DEFINITION 3.2. **[All-provenance]** Given an operator O, input I, output R, and $r \in R$, we define *all-provenance* as $P_{all}(r) = M_{all}(O, I, r \in R)$. □

In many cases P_{all} may be intractable to compute or store, and we may need to resort to "approximations" of it, presented shortly. Alternatively, we may want to find any one (or k) MISets.

DEFINITION 3.3. **[Any-provenance]** Given integer $k > 0$, operator O, input I, output R, and $r \in R$, we define *any-provenance* as any $P_{any}(r) \subseteq P_{all}(r)$ of size $\min(k, |P_{all}(r)|)$. □

Impact-provenance: Given restricted amount of editorial resources, we may want to explore the most *impactful*, i.e., top-l input records sorted by their impact instead of any-k MISets. Our next definition of provenance ranks tuples based on their expected impact on the output record, measured by the number of MISets in which a tuple is present.

DEFINITION 3.4. **[Impact-provenance]** Given an operator O, input I, output R, and $r \in R$, we define *impact-provenance* as $P_{imp}(r) = \{(i, c_i) | \exists M \in P_{all}, i \in M, c_i = \sum_{M \in P_{all}, i \in M} 1\}$. □

Union- and Intersection-provenance: Our next goal is to summarize P_{all} using two approximations: (1) We obtain an "upper bound" provenance that captures the set of all possible inputs *possible* for r, instead of exact combinations of inputs. Therefore, we define the union-provenance of r to be the union of all its MISets. (2) We obtain a "lower bound" provenance that captures the set of all possible inputs *necessary* for r; we define the intersection-provenance of r to be the common input records among all MISets.

DEFINITION 3.5. **[Union-Provenance]** Given an operator O, input I, output R, and $r \in R$, we define *union-provenance* as $P_{uni}(o) = \bigcup_{I_s \in M_{all}(O,I,r \in R)} I_s$. □

DEFINITION 3.6. **[Intersection-Provenance]** Given an operator O, input I, output R, and $r \in R$, we define *intersection-provenance* as $P_{int}(o) = \bigcap_{I_s \in M_{all}(O,I,r \in R)} I_s$. □

It can be seen easily that for operators with unique MISets, P_{uni} and P_{int} coincide.

4. INFERRING PROVENANCE

We now turn to the critical task of deriving the provenance formalisms proposed in Section 3.2. We primarily consider the generic black-box operator case while rigorously examining various properties described in Section 2, and our carry over for more specific cases (e.g., exact or I/O specfications); however whenever necessary we will point out the differences. Table 1 summarizes the results on provenance inference achieved in this section, with details in the following subsections. All time complexities in the table are given in terms of the size of the input N and the size of the output M. The table gives time complexity assuming an operator can be executed freely (i.e., in $\mathcal{O}(1)$); depending on the running time of the operator, the appropriate factor can be multiplied.

Require: $O, I, r \in O(I)$
1: Find an MISet M (Algorithm 4.3)
2: **for** $m \in M$ **do**
3: **if** $r \in O(I - \{m\})$ **then**
4: RETURN "Non-unique"
5: **end if**
6: **end for** RETURN "Unique"

Algorithm 1: Testing the uniqueness of MISets for any monotonic operator.

4.1 Unique MISet Operator

We begin with the case of when a combination of input I, output R, and operator O has a unique MISet for each output record. Note that we don't assume that we know the uniqueness of MISets; instead, we only need the *existence* of a unique MISet. (That is, our results hold even in the case when we don't have any information about the black-box operator, but it just happens that the operator functions in a way that creates a unique MISet for output records.) We have the following main result for unique MISet operators. (Complete proofs for all results in the paper are presented in Appendix A.)

THEOREM 4.1 (COMPUTE MISET). *Given any monotonic operator O, input I, and output R, if O has a unique MISet for each output record $r \in R$, then the unique $MISet$ for r can be constructed in $\mathcal{O}(N)$.*

As a consequence, all entries in Table 1 for unique MISet operators can be solved in $\mathcal{O}(N)$. The above result follows from: (a) the following lemma that tests for uniqueness of MISets; and (b) the fact that a single MISet for any monotonic operator can be computed efficiently using Algorithm 4.3 (Lemma 4.2 presents a more general result for k MISets computation shortly).

LEMMA 4.1. [Uniqueness Test] Given any monotonic operator O, input I, output R, and any $r \in R$, Algorithm 1 tests in $\mathcal{O}(N)$ whether there is a unique MISet for r.

The algorithm works in two stages. First, it finds any MISet M for r. Second, it attempts to find other sets that might produce r by applying O on the entire input except one record from M. If none of these sets produce r, there is no other MISet.

4.2 One-to-One and One-to-Many Operators

For the relatively simple cases of `one-to-one` and `one-to-many` operators, we can obtain easily our composite provenances in linear time in the size of the input N for `one-to-one` operators and linear time in $N + M$ for `one-to-many` operators, as shown by the following theorem.

THEOREM 4.2. Given a `one-to-one` operator O, input I of size N, and an output record $r \in R$, each of $P_{all}(r)$, $P_{any}(r)$, $P_{uni}(r)$, $P_{int}(r)$, and $P_{imp}(r)$ can be computed in $\mathcal{O}(N)$. For a `one-to-many` operator, the complexity is increased to $\mathcal{O}(N + M)$, where $M = |O(I)|$.

4.3 Many-to-one and Arbitrary Operators

Require: $O, I, r \in O(I)$
1: Set $M = I$
2: **for** $m \in M$ **do**
3: **if** $r \in O(M - \{m\})$ **then**
4: M=M-{m}
5: **end if**
6: **end for** RETURN M

Algorithm 2: Computing a single MISet for any monotonic operator.

Require: $O, I, r \in O(I)$, p MISets $\{M_1, \ldots, M_p\}$
1: **for** $(m_1, \ldots, m_p) \in M_1 \times \ldots \times M_p$ **do**
2: Set $I' = I - \{m_1, \ldots, m_p\}$
3: **if** $r \in O(I')$ **then**
4: RETURN Algorithm 4.3 result using O, I', r as input.
5: **end if**
6: **end for**
7: RETURN "No other MISet"

Algorithm 3: Algorithm for finding an MISet different from a given set of p MISets. The algorithm can be applied multiple times to generate several distinct MISets.

Computing P_{any}: We can always find an MISet using an $\mathcal{O}(N)$ algorithm for *any* monotonic operator O. Algorithm 4.3 describes how to find such an MISet. Algorithm 3 provides an extension that finds $k > 0$ MISets (when k MISets exist): For brevity, we specify the algorithm to find a different $(p+1)$th MISet given p MISets. The algorithm is invoked $(k-1)$ times after Algorithm 4.3 successively adding MISets to obtain k distinct MISets. The following lemma establishes our result.

LEMMA 4.2. Given any monotonic operator O, input I, and output record $r \in O(I)$, P_{any} for k MISets can be computed in $\mathcal{O}(N^{k+1})$.

Note that the actual complexity is $\mathcal{O}(N\alpha^k)$ where α is the size of the largest MISet. Therefore, if all MISets are small, the algorithm runs very efficiently.

Computing P_{int}: P_{int} for *any* arbitrary operator can be computed using an $\mathcal{O}(N)$ algorithm. Algorithm 4 shows how to obtain P_{int} and the theorem below establishes the result.

THEOREM 4.3 (P_{int} COMPUTATION). *Given any monotonic operator O, input I, and output record $r \in O(I)$, Algorithm 4 correctly computes the $P_{int}(r)$ with $\mathcal{O}(M+N)$ executions of O.*

Computing P_{uni}: To compute the P_{uni} of an output record $r \in O(I)$, for each input record $i \in I$, we need to determine whether there exists any MISet M containing i. We employ a simple approach to determining if there exists any MISet with i. We simply find all MISets (using Algorithms 4.3 and 3), and determine their union. Note that in the worst case, our naive algorithm takes exponential time in $|I|$. Finding better upper or matching lower bounds is an open problem.

Computing P_{all}: For P_{all}, we show that finding P_{all} for an `many-to-one` monotonic operator is #P-complete[2] in the size of the input; i.e., there does not exists any polynomial-time algorithm to compute P_{all} exactly. Our result is proved using a reduction from the problem of finding all *Minimal Vertex-Cover* in an undirected graph.

THEOREM 4.4. Given any `arbitrary` or `many-to-one` monotonic operator O, input I of size N, output R of size M, and an output $r \in R$, it is #P-complete in N and M to compute P_{all}.

Shortly, we show that the complexity can be made PTIME by restricting ourselves to MISets of bounded size.

[2]#P-completeness corresponds to the class of hard counting problems.

Require: $O, I, r \in Op(I)$
1: Set $S = \emptyset$
2: **for** $i \in I$ **do**
3: **if** $r \notin O(I - \{i\})$ **then**
4: $S = S \cup \{i\}$
5: **end if**
6: **end for** RETURN S.

Algorithm 4: Computing $P_{int}(r)$ for any monotonic operator O.

Computing P_{imp}: Finally, we show that the hardness result P_{all} can be extended easily to P_{imp}.

COROLLARY 4.1. Given any `arbitrary` or `many-to-one` monotonic operator O, input I of size N, output R of size M, and an output $r \in R$, it is #P-complete in N and M to compute P_{imp}.

4.3.1 Bounded-size MISets

Given the intractability of the general problem for P_{all} and P_{imp} for `many-to-one` and `arbitrary` monotonic operators, we explore an intuitive tractable subclass. We consider a practical special case of all MISets being of small size (i.e., bounded by a constant). The following theorem shows that we can now infer all types of P in polynomial time, using an explicit search.

THEOREM 4.5. Given any monotonic operator O, input I, and output $r \in R$ we can find each of P_{all}, P_{any}, P_{uni}, P_{int}, and P_{imp} for r in $\sim N^B$, when for every $S \in M_{all}(r)$, we have $|S| \leq B$.

5. PUTTING IT ALL TOGETHER

5.1 Composing Operator Provenance

So far, we focused only on computing provenance for a single operator. We now consider composition of provenance from single operators into provenance for a chain of operators. Our goal is to understand to what extent (if at all) we can use each individual operators' provenance to determine the provenance of a pipeline. Formally, we would like to solve the following problem.

PROBLEM 5.1. Given monotonic operators O_1, O_2, input I_1, outputs $R_1 = O_1(I_1)$ and $R_2 = O_2(R_1)$, and $r_2 \in R_2$, can we compute $P_*(O_2 \circ O_1, I_1, r_2 \in R_2)$ from $P_*(O_2, R_1, r_2 \in R_2)$ and $P_*(O_1, I_1, r_1 \in R_1)$.

Intuitively, we are interested in generating all provenance P_* for the composition operator $O_{12} = (O_2 \circ O_1)$ from all the provenance P_* of each of O_1 and O_2. Before proceeding to solve the above problem, we make two observations. First, note that Problem 5.1 could have been equivalently defined if O_1 and O_2 were themselves chains of operations with P_* being the provenance of these chains. Our algorithms in Section 4, and hence results in this section, make no assumption on O_1 and O_2 being single operators, so all our results carry over when they are chains of operators. Second, our goal is to explore to what extent the provenance of O_1 and O_2 can be reused to generate the provenance of O_{12}, without any additional execution of O_1 or O_2.

Our first main result shows that the execution of $O_2 \circ O_1$ can be completely simulated using P_{all} for O_1 and O_2, and hence all provenance of $O_2 \circ O_2$ can be computed as in Section 4. The theorem gives a constructive algorithm, and hinges on the core idea that P_{all} for any monotonic operator captures enough information to execute the operator on any subset of its input.

THEOREM 5.1. Given monotonic operator O_1, O_2, input I_1, outputs $R_1 = O_1(I_1)$ and $R_2 = O_2(R_1)$, and $r_2 \in R_2$, for any $I_s \subseteq I_1$, we have $r_2 \in (O_2 \circ O_1)(I_s)$ if and only if

Properties	$P_*(O_2 \circ O_1,)$
O_1: arbitrary O_2: arbitrary	${P_{uni}}^{12}(r_2) \subseteq \bigcup_{r_1 \in {P_{uni}}^2(r_2)}({P_{uni}}^1(r_1))$ ${P_{int}}^{12}(r_2) \supseteq \bigcup_{r_1 \in {P_{int}}^2(r_2)}({P_{int}}^1(r_1))$
O_1: one-to-one O_2: arbitrary	${P_{all}}^{12}(r_2) = \{\bigcup_{s1 \in s2} {P_{any}}^1(s_1) \vert s2 \in {P_{all}}^2(r_2)\}$ $P_*^{12}(r_2) = \bigcup_{s1 \in P_*^2(r_2)} {P_{any}}^1(s_1)^\dagger$
O_1: arbitrary O_2: one-to-one	$P_*^{12}(r_2) = P_*^1(P_*^2(r_2))^\ddagger$

Table 2: Problem 5.1 for combinations of properties for O_1 and O_2 operating on inputs I_1 and $R_1 = O_1(I_1)$ respectively, and R_2 is the result of O_2. The table uses $P_*^{12}(r_2)$, $P_*^2(r_2)$, and $P_*^1(r_1)$ as shorthands for $P_*(O_2 \circ O_1, I_1, r_2 \in R_2)$, $P_*(O_2, R_1, r_2 \in R_2)$ and $P_*(O_1, I_1, r_1 \in R_1)$ respectively. † * stands for one of *uni*, *int*, and *any*. ‡ We have slightly abused notation to apply P_*^1 to a singleton set, instead of the record in the set itself.

$\exists M_2 \in P_{all}(O_2, R_1, r_2)$ such that $M_2 \subseteq O_s$, where $O_s = \{r_1 \vert \exists M \in P_{all}(O_1, I_1, r_1 \in R_1)\ s.t.\ M \subseteq I_s\}$.

Given the above result, we know that P_{all} for O_1 and O_2 contain enough information to compute all P^* for $O_2 \circ O_1$. However, using P_{all} can be expensive because of the number of possible MISets. Hence, our next goal is to attempt to use other forms of provenance of O_1 and O_2, i.e., without enumerating P_{all}. If some provenance of O_{12} cannot be computed directly, we can fall back on the techniques from Section 4 to generate the provenance or use P_{all} using Theorem 5.1.

Next we look at special cases of operators and determine when P_* for $O_2 \circ O_1$ can be computed efficiently. Table 2 summarizes our results. The table is *complete* in the sense that for any P_* not present in the table, we must use P_{all} for O_1 and O_2 (using Theorem 5.1) to compute the entry, or resort to techniques in Section 4. Given all possible combinations of operator properties is too many to list (16 combinations of arbitrary, `many-to-one`, `one-to-one`, and `one-to-many`), Table 2 presents a delineating subset of results. All other combinations of results can be derived from the entries in the table. For instance, when O_2 is `one-to-one` P_*^{12} can be computed for arbitrary O_1, hence other combinations involving O_1 aren't present in the table. Similarly, we only consider arbitrary O_2 when O_1 is arbitrary. Further, we do not consider `many-to-one` separately as results for `many-to-one` and arbitrary are similar. Also, results for `one-to-one`, `one-to-many` are similar as they both ensure a unique singleton MISet for each output record. We don't separately consider unique MISets, as all of P_{all}, P_{any}, P_{uni}, P_{int} are the same, and computed easily. Finally, the table omits P_{imp} as our solution for P_{imp} is equivalent to that of using P_{all}.

5.2 Properties and Provenance Selection

To summarize, given an IE execution our approach, PROBER, allows users to specify the output records that they are interested in debugging. Either using information such as IO specifications or integrity constraints or using sampling, PROBER attempts to identify the type of the operators (e.g., `one-to-one`, or arbitrary). In the absence of any conclusive information, PROBER assumes an arbitrary operator. For each operator or pipeline, users may choose the type of provenance they want based on editorial resources available. Note that users may always start with the conservative P_{int} or P_{any}, and explore more complex provenance, such as P_{all} as needed, or ask for input to be ranked, such as P_{imp}. We make two important observations regarding this user exploration: (1) Whenever operators satisfy restricted properties (such as `one-to-one`), PROBER readily computes all forms of provenance very efficiently. (2) For arbitrary monotonic operators, all our algorithms proceed in a "pay-as-you-go fashion"; for instance, even if a user would like to perform an in-depth analysis of P_{all} leading to a potentially expensive computation, PROBER starts returning MISets immediately and progressively provides more information as available. Specifically, our P_{any} algorithm keeps iteratively finding new MISets, which are returned to users as found.

6. EXPERIMENTAL EVALUATION

We now present results from our experimental evaluation. After describing our data sets (Section 6.1), we present a qualitative study of PROBER (Section 6.2). Next, we evaluate the effectiveness of our basic unit for provenance, namely, MISets (Section 6.3). Then, we perform a detailed study of various provenance formalisms (e.g., $P_{uni}, P_{int}, P_{any}$, etc.) by discussing basic statistics (Section 6.4), and then, compare their coverage (Section 6.5) and execution times (Section 6.6).

6.1 Experimental Settings

Data sources: We used a collection of 500 million web pages crawled by the Yahoo! search engine.

Extraction pipelines: For our IE tasks, we implemented two pipelines. Our first pipeline, denoted, `Bussiness`, is as described in Section 2. For our second pipeline, denoted `Iterative`, we reimplemented a state-of-the-art bootstrapping exraction technique described by Pasca et al. [12] for large-scale datasets such as Web corpora which is similar in spirit to other IE pipelines such as Snowball [1] and Espresso [13].

Extracted relations: As extraction tasks, we focus on six relations (the last column shows the number of extracted tuples):

1	`footwear`:	⟨name, address, phone⟩	340,131
2	`actors`:	⟨movie, actor⟩	14,414
3	`books`:	⟨book, author⟩	142,337
4	`mayor`:	⟨U.S. city, mayor⟩	28514
5	`sen-party`:	⟨senator, affiliated party⟩	2,119
6	`sen-state`:	⟨senator, state⟩	14,582

We built `footwear` using `Bussiness` using a corpus of 5,443,183 web pages from 147 sites (see Section 2); all the other relations were built using `Iterative`. Our qualitative analysis presented next is using the `footwear` dataset. The empirical analysis that follows was performed on each of `actors`, `books`, `mayor`, `sen-party`, and `sen-state`. For most of our experiments we show results for the high-confidence tuples from our datasets, as these results are the most interesting: High-confidence tuples have most number of contributing input records and are therefore the hardest for provenance and debugging.

6.2 Qualitative analysis of PROBER's utility

To gain insights into the utility of PROBER, we performed a qualitative analysis of the records generated for `footwear`. Among the final set of output records, 38% were missing business names, 40% were missing phone numbers, 37% were missing addresses. To give a flavour of user interaction with PROBER, we qualitatively depict a debugging analysis for a specific erroneous record. In particular, we explore a record, r, ⟨'AUSTIN, TX', 'Burnett St, Austin, Texas 78703', null⟩ which has incorrect values for business name and a missing value for phone number. Through the source web page associated with this record, we found that our first operator, namely `sg`, had incorrectly segmented the page. As shown in Figure 2, `sg` generated an incorrect segmentation for the second and third business contacts listed on the page. By fixing this segmentation, we debug and correct record r as well as other records extracted from this page.

6.3 Is MISet an effective representation?

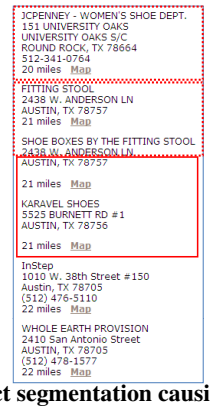


Figure 2: Incorrect segmentation causing incorrect output.

Earlier in Section 3, we proposed MISets as the primary representation to collect information related to an output record for debugging purposes, and provided concrete theoretical justification for our choice. Of course, other representations are also possible in practice. Next, we present an experimental comparison of MISets against three strong baselines that could be used to collect tracing information for an output record. Specifically, we compare the following methods for generating tracing information.

- `All-recs`: The naive baseline of repeatedly exploring all input records for every output record.
- `Wrd-OR`: Using the bag of words in an output record, we build a keyword query to fetch all input documents containing at least one term using a standard IR-like search interface.
- `Wrd-AND`: Similar to `Wrd-OR`, except we only fetch documents containing *all* terms in the output record.

An important note about the `Wrd-OR` and `Wrd-AND` baselines is that they exploit specific information about the extraction operators, namely, that input records aren't "mangled", i.e., terms are preserved by the extraction. MISets, on the other hand, use no such information. Since in our extraction scenario, we chose operators that do indeed preserve terms in records, our comparison is unfair in that it favors `Wrd-OR` and `Wrd-AND`. Our goal was to compare MISets with the *best* possible scenario for our baselines. (Clearly, in a fair comparison including operators that generate new terms or alter terms in input records, `Wrd-OR` and `Wrd-AND` won't even be applicable, and `All-recs` would be the only feasible baseline.)

Figure 3(a) presents our results comparing each method (MISets, `Wrd-AND`, `Wrd-OR`, `All-recs`) by examining the total number of input records that need to be fetched in order to generate the tracing information, varying the number of output records. By design, `Wrd-AND` retrieves the fewest possible documents and MISets completely coincides with `Wrd-AND`. Indeed, this "experimentally proves" our claim from Section 3 that MISets retrieve minimal sets of records from the input. Note that even in our favorable setting for keyword-based retrieval, `Wrd-OR` retrieves many more input records[3], and `All-recs` is even more prohibitively expensive.

6.4 Size of provenance formalisms

Next we explore the size of provenance generated using each of our formalisms: $P_{all}, P_{uni}, P_{int}$, and P_{any} with $k = 1, 3, 5$. (P_{imp} isn't shown as the size of P_{imp} is naturally equal to the number of input records requested.) Figure 3(b) shows the average size of the provenance generated for each of the provenance formalisms over a set of 50 tuples ranked by their confidence scores. It is noteworthy that the sizes of the provenances, and in turn, the manual effort necessary can substantially vary across tuples. Since P_{all} maintains all possible MISets, it is the largest. From the figure, we learn that a practical choice for users would be to start exploring P_{int} or P_{any}-1, then request P_{any}-k for $k > 1$ and P_{uni} if necessary.

To gain more insight into the distribution of sizes for individual tuples, Figure 3(c) plots the size of each provenance type for the top-30 tuples. The size of P_{all} varies significantly but is almost always significantly more than all other provenance types. The two cases where P_{all} coincides with other provenance types are examples of output records with unique MISets. The minor variations in the sizes of all other forms of provenance are obscured by the log-scale for the y-axis.

6.5 Coverage

Next we explore the *coverage* of each provenance model measured as $\frac{|P_*|}{|P_{uni}|}$. Our goal is to determine what fraction of all potentially contributing input records is retrieved by any single MISet or any arbitrary 3 or 5 MISets, as well as by P_{int}. Figure 3(d) shows the coverage averaged over the set of output records. P_{int} has very low coverage indicating that very few input records are *essential* in producing any output record; in other words, in most cases there are many different explanations for the same output record. P_{any} (for $k = 1, 2, 3$), on the other hand, retrieves a sizable fraction of all contributing input records. This indicates that using P_{any} is a practical solution to start debugging, by retrieving the initial set of input records, and if necessary request more MISets.

We treat the coverage study for P_{imp} as a special case. Since the coverage of P_{imp} depends on the number of ranked tuples retrieved, we measure the coverage of top-k P_{imp} records $\{r_1, \ldots, r_k\}$ using two measures: (1) Record-coverage measured as the fraction of the total number of record appearances of these k records in P_{all}. That is $\frac{\sum_{i=1}^{k} c_i}{\sum_{i=1}^{l} c_i}$, where c_i denotes the number of MISets containing r_i, and P_{all} contains records $\{r_1, \ldots, r_l\}$. (2) MISet-coverage measuring the fraction of the total number of MISets that contain some tuple in $\{r_1, \ldots, r_k\}$. Figure 3(e) shows these coverages for P_{imp}; we observe that P_{imp} is very effective in giving very high MISet-coverage with very few retrieved records, justifying that retrieving few tuples from P_{imp} can be very useful in debugging with a high representation of almost all MISets. We get high incremental value for initial records, with diminishing returns as we retrieve more tuples. For record-coverage the trend is closer to a linear increase in coverage. Overall, we observe that P_{imp} (along with with P_{any} and P_{int}) can be effective tools for debugging, with the caveat that P_{imp} is computationally more expensive (see Section 6.6). An interesting open question arising is that of efficiently (to the extent possible, given our #P-complete result from Section 4) retrieving just sufficient number of records to meet a coverage demand.

6.6 Build time

Finally, we study the time required to build provenance in PROBER, which directly depends on the amount of data fetched. Figure 3(f) plots the number of input records fetched for each type of provenance, varying the number of high-confidence records. We note that P_{all}, P_{uni}, and P_{imp} are the most expensive computationally, while the amount of data fetched for P_{int} and P_{any} for $k = 1, 3, 5$ is significantly less. Since P_{all} requires a large amount of data to be fetched, we studied the behavior of our algorithm for finding all MISets when the size of each MISet is bounded below 5 (Section 4.3.1). We notice that this is more expensive than P_{any} and P_{int} but significantly faster than P_{all}, and hence information

[3]Note the log-scale on the y-axis

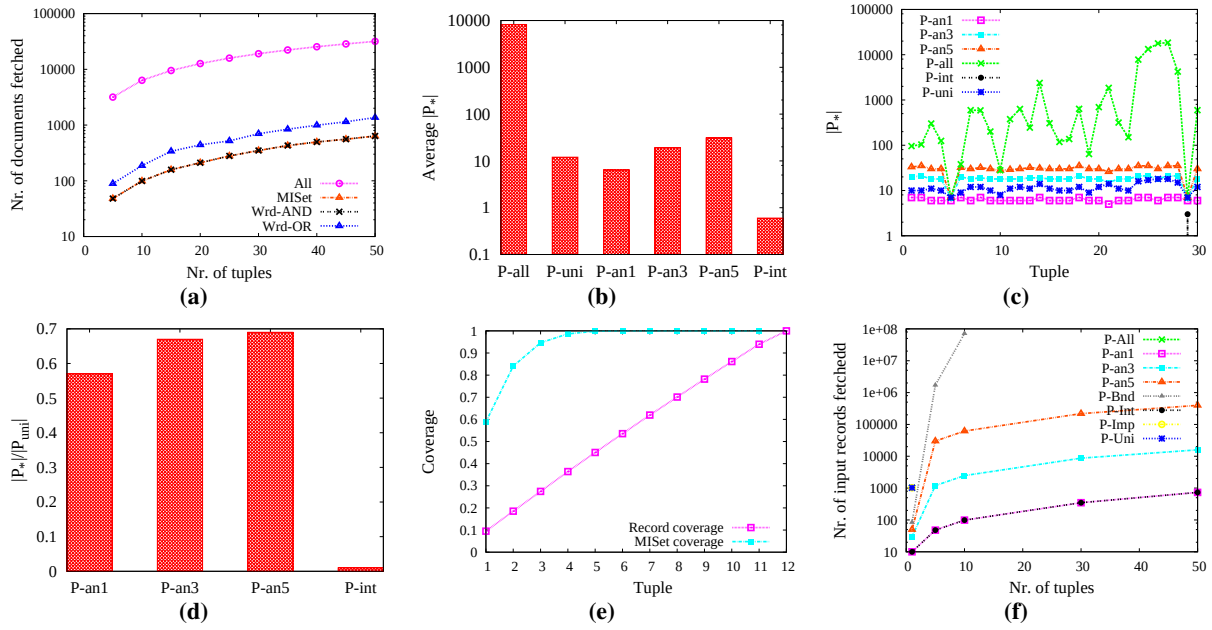


Figure 3: **(a) Number of documents fetched representing the amount of work necessary when using different debugging paradigms. (b) Average size of various provenances over 50 tuples. (c) Size of provenance generated for top-30 tuples ranked by confidence scores. (d) Coverage of different provenances with respect to P_{uni}. (e) Coverage of P_{imp} with respect to total number of MISets, and total contribution of all input records. (f) Data fetched to derive various provenance P_* definitions.**

on the size of each MISet can potentially be useful.

6.7 Evaluation summary

In conclusion, we established the utility of PROBER over a variety of relations. MISets pick out minimal sets of input records in comparison to other baseline methods thus enabling rapid resolution of output records. Our provenance formalisms may substantially vary in their sizes and we discussed how users may gradually move from exploratory provenances to more complete ones. Finally, we studied the tradeoff between coverage and execution time for various provenance formalisms.

7. RELATED WORK AND CONCLUSIONS

Here we present a very brief discussion of related work, with a more comprehensive description appearing in Appendix B. Some recent work [6, 7, 8, 9, 16] has broadly looked at providing *exploration* phases that enable users to determine if a text database is appropriate for an IE task. However, users are provided with little or no insight into why unexpected results are produced, and how to debug them. Another interesting piece of work [15] presented techniques to build IE programs using Datalog for greater readability and easier debugging. Our recent work [14] considered debugging for iterative IE, and [4] looked at provenance for *non-answers* in results of extracted data. However, these papers assume complete knowledge of each operator in some form, such as access to code for each operator, or SQL queries applied to input data. Finally, there is a large body of work on provenance for relational data (refer [5, 17]), and more recently [3] on understanding provenance information. This work does not address the problem of building provenance for black-box operators to facilitate IE debugging with minimal editorial effort, the primary goal of our work.

In conclusion, we presented PROBER, the first system for ad-hoc debugging of IE pipelines. At the core of PROBER, is a suitable MISet-based provenance-model to link each output record with a *minimal* set of contributing input records. We provided efficient algorithms and complexity results for provenance inference, and an extensive experimental evaluation on several real-world data sets demonstrating the effectiveness of PROBER. A few specific directions for future work arise, such as tighter bounds for P_{uni} inference, and extending to non-monotonic operators. A more general direction of future work we are currently pursuing is to develop an interactive GUI for PROBER and perform a user study by deploying it for multiple extraction frameworks at Yahoo!.

8. REFERENCES

[1] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *DL*, 2000.

[2] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[3] A. Chapman and H. V. Jagadish. Understanding provenance black boxes. *Distributed and Parallel Databases*, 27(2), Apr. 2010.

[4] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1), 2008.

[5] R. Ikeda and J. Widom. Data lineage: A survey. Technical report, Stanford University, 2009.

[6] A. Jain, A. Doan, and L. Gravano. Optimizing SQL queries over text databases. In *ICDE*, 2008.

[7] A. Jain and P. G. Ipeirotis. A quality-aware optimizer for information extraction. *ACM Transactions on Database Systems*, 2009.

[8] A. Jain, P. G. Ipeirotis, A. Doan, and L. Gravano. Join optimization of information extraction output: Quality matters! Technical Report CeDER-08-04, New York University, 2008.

[9] A. Jain and D. Srivastava. Exploring a few good tuples from text databases. In *ICDE*, 2009.

[10] G. Kasneci, S. Elbassuoni, and G. Weikum. Ming: mining informative entity relationship subgraphs. In *CIKM*, 2009.

[11] M. Paşca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Names and similarities on the web: Fact extraction in the fast lane. In *Proceedings of ACL*, July 2006.

[12] M. Paşca, D. Lin, J. Bigham, A. Lifchits, and A. Jain. Organizing and searching the world wide web of facts - step one: The one-million fact extraction challenge. In *Proceedings of AAAI-06*, 2006.

[13] P. Pantel and M. Pennacchiotti. Espresso: leveraging generic patterns for automatically harvesting semantic relations. In *Proc. of ACL*, 2006.

[14] A. D. Sarma, A. Jain, and D. Srivastava. I4E: Interactive investigation of iterative information extraction. In *SIGMOD*, 2010.

[15] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. 2007.

[16] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Towards best-effort information extraction. In *SIGMOD*, 2008.

[17] W.-C. Tan. Provenance in Databases: Past, Current, and Future. *IEEE Data Engineering Bulletin*, 2008.

APPENDIX

A. PROOFS

Proofs of Theorem 4.1, Lemma 4.1, and Lemma 4.2: To prove Lemma 4.1, consider Algorithm 1, which attempts to find non-uniqeueness of MISets, starting from a given MISet M obtained from Algorithm 4.3. Any other MISet M' cannot be a superset of M (else it wouldn't be minimal). Therefore, there must exists some $m \in M$ and $m \notin M'$, which implies that $I - \{m\} \supseteq M'$. Therefore $O(I - \{m\})$ must contain r for some $m \in M$ if there exists a MISet other than M.

The basic algorithm for Lemma 4.2 is Algorithm 4.3 which finds any MISet for a given input I. Finding k MISets is simply obtained by modifying input I and calling Algorithm 4.3 recursively: To find an MISet different from M, for each element $m \in M$, Algorithm 4.3 is called with $I - \{m\}$. Similarly, given p MISets $M_1, \ldots, M_p$, to find a $(p+1)$th MISet M', M' must differ from each of $M_1, \ldots, M_p$. Hence, there must exist a p-tuple $(m_1, \ldots, m_p)$, $m_i \in M_i$, such that $M' \subseteq I - \{m_1, \ldots, m_p\}$. Our algorithm attempts to find an MISet for every such p-tuple. Finding the $(p+1)$th MISet needs to iterate over $|M_1| \cdot \ldots \cdot |M_p|$ p-tuples in the worst case, giving us the required complexity.

Theorem 4.1 follows based on checking whether O gives a unique MISet (Lemma 4.1), then using Lemma 4.2 with $k = 1$. □

Proof of Theorem 4.2: We scan the input records $i \in I$, one at a time, and apply O to $\{i\}$ individually. Whenever we have $O(\{i\})$, we return $P_{any}(o) = \{i\}$, and add $\{i\}$ to $P_{all}(o)$, and add the element i to $P_{uni}(o)$, initialized to $\emptyset$. If $|P_{uni}(o)| \geq 2$, we set $P_{imp}(o) = \emptyset$, else set $P_{imp}(o) = P_{uni}(o)$. Finally, P_{int} can be computed from P_{uni}.

It can be seen easily that provenance for `one-to-many` operators can be computed in a similar fashion. The only difference is that the number of output records can now be larger than the number of input records, i.e., M may be larger than N. Hence the complexity increases to $\mathcal{O}(M + N)$. □

Proof of Theorem 4.3: We perform $\mathcal{O}(N)$ calls to the operator, and for each call, we may have to look at an output of size $\mathcal{O}(M)$. Correctness of the algorithm follows easily: For any record i to be in the intersection of all MISets, removing i from the input must remove the output record. □

Proof of Theorem 4.4: First we prove #P-hardness for a `many-to-one` operator (and hence for an `arbitrary` operator), and then show that the problem is in $\#P$, which applies for `many-to-one` and `arbitrary` monotonic operators, completing our proof.

1. **#P-hardness** We give a reduction from the problem finding all minimal vertex covers. Given a graph $G(V, E)$, our goal is to compute all $V_{min} \subseteq V$ such that (1) V_{min} is a *cover*: each edge $e \in E$ has an endpoint in V_{min}, (2) V_{min} is *minimal*: No proper subset of V_{min} is a cover. Given the input $G(V, E)$, we create an instance of finding P_{all} as follows: $I = V$, $O = \{1\}$, our goal is to find all MISets of 1. O takes as input any subset $I_s \subseteq I$, and returns $\{1\}$ if the corresponding set of vertices V_s is a (not necessarily minimal) cover of E in G, and returns $\{0\}$ otherwise. Note that each minimal vertex cover of G corresponds to a MISet of 1, and each MISet gives a minimal vertex cover. Finally, note that our operator generates a single output record, and is therefore `many-to-one`.
2. **#P** Given any $I_s \subseteq I$, we can check in PTIME whether I_s is an MISet: I_s is an MISet if and only if no subset of it obtained by removing a single element returns $\{1\}$, and $1 \in Op(I_s)$. Therefore, we can check for all sets in any P_{all}, whether each of them is an MISet. □

Proof of Corollary 4.1: Note that the hardness result of Theorem 4.4 holds even if our goal was to only count the number of minimal vertex covers, or equivalently, find the number of MISets. We can translate the problem of counting the number of MISets to computing P_{imp} for a special input tuple $i^* \in I$. Given an input $(Op, I, o \in O)$ to P_{all}, we create $(Op', I', o \in O)$, where $I' = I \cup \{i^*\}$ and for any $I_s \subseteq I'$ we have $Op'(I_s) = Op(I_s)$ if and only if $i^* \in I_s$ and $Op'(I_s) = \emptyset$ if $i^* \notin I_s$. Counting the number of MISets for O now reduces to the problem of determining P_{imp} for i^*. □

Proof of Theorem 4.5: The theorem follows directly based on an explicit search over all possible inputs of size of at most B to find P_{all}. All other P_* are subsequently computed using P_{all}. □

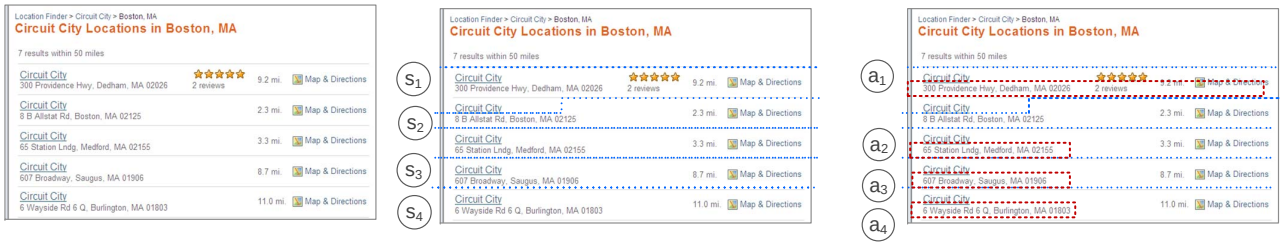
Proof of Theorem 5.1: The main idea used in the result is that the property of MISets for monotonic operators ensures that $\forall r$: $P_{all}(O, I, r \in R)$ for any operator is sufficient to reconstruct (and execute) O for any subset $I_s \subseteq I$: Using monotonicity, we know that $O(I_s) \subseteq O(I)$, hence we only need to determine for every $r \in R$, whether $r \in O(I_s)$. Using the property of MISets, we have $r \in O(I_s)$ if and only if there is a MISet of r, say $M_r \subseteq I_s$, allowing us to exactly construct $O(I_s)$.

Given the above fact that P_{all} enables reconstructing any operator, the two expressions in the theorem merely simulate the execution of each operator: For a record r_2 to be in the output of $(O_2 \circ O_1)$, some MISet M_2 of r_2 for O_2 must be contained in the output of O_1. Such an MISet M_2 is in the output of O_1, i.e., $M_2 \subset O_s$. □

B. EXPANDED RELATED WORK

Recognizing the need for a principled approach to assisting IE developers and users, several methods have been proposed to enable *exploration* phases. Shen et al. [16] presented an iterative approach to developing IE systems, where users begin with an "approximate" extraction query. Based on the results of this query, users may refine the follow-up query. Jain et al. [9] presented a query model for IE tasks for the purpose of exploring whether a database is useful for the IE task or not. Following a similar spirit, optimization strategies that enable users to efficiently fetch IE results with pre-specified output quality (e.g., minimum number of good tuples and maximum tolerable bad tuples) have been proposed for single IE systems [6, 7] as well as multiple IE systems [8]. While such exploration phases enable IE developers to assess the quality of an IE system, they mostly focus on answering the question, "*Is a text database D a good choice for the IE system at hand?*" Furthermore, users are provided with very little insights into why an IE system does not perform as expected.

Assuming full access and control to the code for each operator in an IE pipeline, prior work [15] presented methods to build IE programs involving multiple operators using Datalog to generate programs that are easy to read and thus easier to debug. Our approach considers a generic IE pipeline that may involve any arbitrary operators for which we may not have exact specifications or access to the code. Recently, [3] addressed the problem of understanding "provenance black boxes"; the goal of their work is to provide users with an easier way to understand provenance information, allowing them to aggregate or drill down on provenance. In contrast, our goal is to build a provenance model that is suit-



Input web page

Segments identified on the page

Addresses identified on the page

Figure 4: Sample input output for three steps, namely, `sg`, `ad`, and `pn`, in our extraction pipeline.

able for black-box operators in an extraction pipeline. Note that we make no direct contribution on user understanding of provenance; rather, we produce *minimal sets* of provenance and their compositions in order to quickly understand errors in the data produced by the pipeline.

Our prior work [14] presented debugging algorithms for IE tasks; our current work substantially differs from and extends this work. The techniques in [14] focused on a simple form of IE system, namely, iterative IE methods [11]. Furthermore, we assumed we had complete knowledge on how each operator was designed and exact operator and input-output specifications . Moreover, the focus of [14] was to utilize the relatively simple provenance model to enable efficient algorithms for explanation, diagnosis, and repair. This paper developed a new provenance model for arbitrary extraction operators, and presented algorithms for building this provenance. Also relevant previous work on provenance is that of [4], which addresses the problem of deriving the provenance (explanations) for *non-answers* in extracted data. The paper considers conjunctive queries, and for every potential tuple t in an answer to a conjunctive query, the authors provide techniques for determining updates to base data that would produce t in the output. Once again, our work relaxes these assumptions and enables debugging over complex IE pipelines consisting of arbitrary black-box operators. Finally, there is a large body of previous work on provenance for relational databases (refer to [17, 5] for surveys); this work does not meet our two-fold requirements of provenance for black-box operators, and designing provenance to minimize editorial work during debugging.

C. EXTENSIONS

In this section, we very briefly discuss the extension of PROBERs framework for absence of records from the output, which is particularly useful for non-monotonic operators. We emphasize that this section is primarily meant to indicate that PROBER is amenable to these extensions. However, we are currently developing precise details, and our current system does not support these extensions.

For debugging the absence of records from an output of any non-monotonic operator, we may analogously define a notion of *Maximal Superset* (MASet).

DEFINITION C.1. [MASet] Given an operator O, its input I and output R, we say that $I_s \subseteq I$ is a *Maximal Superset* (MASet) of $r \notin R$ if and only if: (1) $r \in Op(I_s)$; and (2) $\forall I' : I' \supset I_s, I' \subseteq I \Rightarrow r \notin Op(I')$. □

Just as in the case of MISets, it's easy to see that MASets are also not unique:

EXAMPLE C.1. In Example 3.1, if the operator returned "NO" whenever there were fewer than 50 records in the input, then the MASet of "NO" is any set of 49 input records. □

MISets are useful for debugging based on the *presence* records in the output of an operator, while MASets are useful for debugging the *absence* of records from the output: For every record that is output by an operator, its MASet is the entire input, and hence its MASet doesn't help in fixing an erroneous output record. However, the MISet of an erroneous record points to potential incorrect input that caused the error. Conversely, for a record that is absent in the output, MASets help identify what caused the record to get omitted from the output.